2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

UVM IEEE Shiny Object

Rich Edelman
Mentor, A Siemens Business
46871 Bayside Parkway
Fremont, CA 94538

Moses Satyasekaran
Mentor, A Siemens Business
46871 Bayside Parkway
Fremont, CA 94538

***Abstract*- The UVM IEEE standard [8] has been published for some time and the first implementation library [7] is released. This paper will explore a focused part of the UVM IEEE. Central core functions will be discussed in terms of the core services class. Worked examples will demonstrate integrating these core functions in UVM-1800.2-2017-1.0.**

I. INTRODUCTION

This paper is part history and background; part performance analysis; part UVM development archeology; part testbench architecture; and part pining for a future of stability and simplification for verification engineers. It shares tips for creating a useful report server, or replacing the factory, or installing a new core service. It also discusses improvements to debug for the configuration database that continue to be needed after many years. There is no surprise ending. The UVM is here to stay, being used widely for verification productivity improvements world-wide. And the UVM-IEEE is the latest release.

II. THE UVM IEEE

The UVM IEEE is substantially the same as UVM 1.2 [6] and UVM 1.1d [5], with just enough backward incompatibilities to keep things interesting. The released code unpacks with a name like 'uvm-core-1800.2-2017-1.0'. I'd prefer uvm-1800.2-2017-1.0 or better and in keeping with uvm-1.1d and uvm-1.2 - uvm-ieee-1.0. In any case, in this paper the release will be called 'UVM IEEE'.

What's New in UVM IEEE?

The README.md file in the release has notes about changes. Comment lines were added throughout the UVM IEEE source code, annotating what is part of the standard, what might be considered for inclusion in the future, and what is in the Accellera implementation (@uvm-ieee, @uvm-contrib and @uvm-accellera, respectively).

Certain UVM 1.2 APIs were deprecated. They are still available behind a switch, but will be removed in the next release. Any code that was previously deprecated in UVM 1.2 has been removed.

There are some short migration instructions (my emphasis, formatting and typo corrections):

1. Compile/Run using a UVM1.2 library **with `*UVM_NO_DEPRECATED*`** defined.
   This will ensure that your code runs with UVM 1.2 which was a baseline for the IEEE 1800.2 standard development.
2. Compile/Run using this library **with `*UVM_ENABLE_DEPRECATED_API*`** defined.
   This step helps identify the areas where your code may need modifications to comply with the standard.
3. Compile/Run using this library **without `*UVM_ENABLE_DEPRECATED_API*`** defined.
   Removing the `define ensures that only the 1800.2 API documented in the standard, along with any non-deprecated Accellera supplied API, is used. Any new compile failures are the result of deprecated 1.2 APIs.

The migration instructions sound simple, but experience suggests otherwise. A testbench which used many of the detailed APIs and tricky techniques in uvm-1.1d or uvm-1.2 will have problems migrating. They are simple problems usually, requiring less than a week to manage. The suggestion for the future remains – use only as much of the UVM as needed, and no more. In that way, changes are less trouble in later releases. The benchmark example in this paper runs unchanged on all three UVM versions (uvm-1.1d, uvm-1.2 and UVM-IEEE). It is simple, but yet a complete testbench.

The DEVIATIONS.md file in the release has notes about issues that were found in the UVM IEEE LRM which were corrected by writing different (non-compliant) code in the UVM IEEE implementation. These are items to be changed in the LRM for a future release. For example, the LRM defined do_kill() as non-virtual. It should be virtual. The LRM defined unlock() and ungrab() as tasks. They should be functions. There are a total of 10 of these kind of items listed.

The docs directory in the release contains a surprisingly small HTML "UVM 1800.2-2017 Class Reference Manual". (uvm-core-1800.2-2017-1.0/docs/html/index.html)

The src directory in the release contains the usual contents. It will be familiar to any UVM user.

What's NOT in UVM IEEE?

The UVM IEEE implementation does not include any examples and has no user manual. It does not include any significant improvement for built-in debug.

III. UVM IEEE SPEED

The UVM is getting slower. These numbers were collected from a trivial testbench with two UVM environments. Each UVM environment generates 10,000,000 transactions and sends them through a sequencer to a driver which simply delays a random amount and prints a message. It does nothing useful.

UVM Version	Elapsed Time (Relative Wall Clock)
UVM-1.1d	100
UVM-1.2	116
UVM-IEEE	136

This benchmark is not a fair benchmark since there is no RTL nor GATES, and it does nothing useful. It does NOT indicate that a UVM IEEE test bench will be 1.36 times slower than UVM 1.1d. But, it does indicate that the basic infrastructure of the UVM has been getting slower with each release. For a small test bench with a small number of transactions, the difference in speed won't be noticeable.

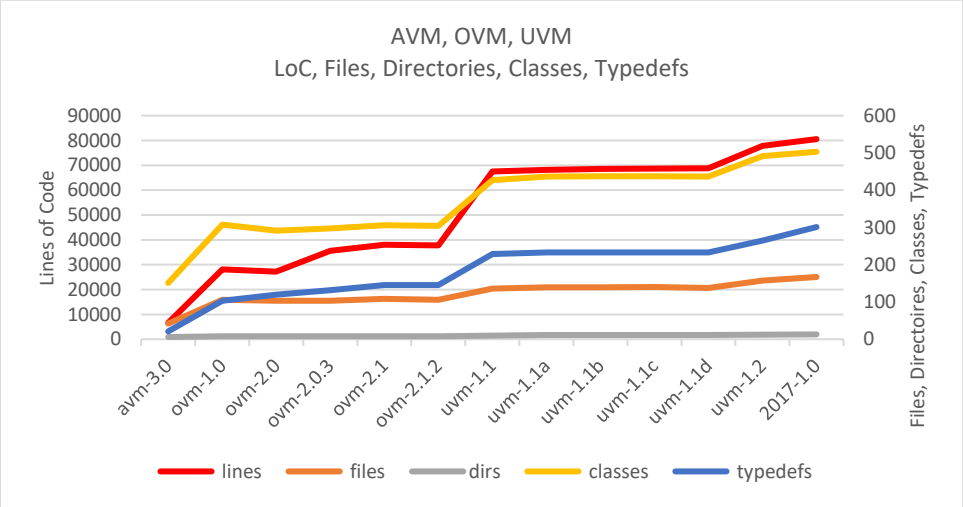
IV. UVM IEEE CODE BASE SIZE

The UVM codebase is increasing in many measures. More files, more classes, more lines, more macros. The software industry has many statistics for "bug rate per lines of code". Essentially, the more lines of code, the more bugs. This is not to say that the UVM IEEE has more bugs than UVM 1.1d, but there are 12,000 additional lines. At

a rate of 10 bugs per 1000 lines of code, that's 120 potential additional bugs. The industry rates vary, but typical rates have been 15-50 bugs per 1000 lines of code.[1]

Name	lines	files	directories	classes	typedefs
avm-3.0	6,608	42	6	151	21
ovm-1.0	28,098	106	8	308	103
ovm-2.0	27,151	103	8	291	120
ovm-2.0.3	35,646	103	8	297	132
ovm-2.1	38,035	108	8	306	145
ovm-2.1.2	37,786	106	8	304	145
uvm-1.1	67,466	136	10	427	229
uvm-1.1a	68,129	139	11	436	233
uvm-1.1b	68,551	139	11	437	233
uvm-1.1c	68,690	140	11	437	233
uvm-1.1d	68,799	138	11	436	233
uvm-1.2	77,781	157	12	491	265
1800.2-2017-1.0	80,549	167	13	503	301

Graphically, the same data:



This data is not meant to be a concrete measure of quality or complexity. It is not meant to imply any lack of sophistication. This data is meant to start a discussion about quality and complexity.

Simpler APIs are required. Easier to understand debug messages are required. Better use-models and documentation is needed. Generally, the goal is to have the UVM adopted. It should be easy to adopt.

Simplicity does not imply unsophisticated. It implies well thought out and easy to explain. It also implies easy to understand.

A very smart verification engineer once said that the tests in a verification environment should be able to be written by someone who has had too much to drink. It is likely that this will not be a goal, but the idea still resonates 12 years

later. Make the interface stable and simple. The UVM IEEE did not become simpler, easier to use or more stable. These should be goals for the next releases.

V. UVM Background

The UVM has roots in many places, with many valued contributors. This paper will not enumerate or list the participants – there are many.

Beginning with the AVM and then the OVM and finally the UVM, the desire was always to provide a simplification and a methodology which would improve productivity for verification engineers. SystemVerilog was a big language, released as 3.1a in 2004. The IEEE released SystemVerilog 1800-2005 through 1800-2017 [4]. The AVM was released in 2007, followed by the OVM in 2008 and finally UVM 1.1 in 2011. The UVM release needed a few additional releases to stabilize and to gain the functionality required and expected by users. The UVM 1.1d version has proven solid and stable, released in 2013. UVM 1.2 was released in 2014. The IEEE became involved and the UVM IEEE Language Reference Manual was released in 2017. In 2018, a sample implementation named uvm-1800.2-2017-1.0 was released.

VI. Digging into UVM IEEE Developments and Changes

There are too many changes in the UVM IEEE for a complete discussion in this short paper. But certain features and changes can be discussed as an introduction or a representative "random" set of changes. Many changes in UVM IEEE are simple comment changes or documentation updates. Some basic concepts changed (policy classes, abstract classes, local and protected variables). These changes could be classified as "plumbing changes". Perhaps improvements, but not required by any user. They are small in scope and likely harmless in the big picture. Some changes are bigger or have potential to cause problems. The number of possible issues could be large.

Changes that cause issues are quite hard to find. They are usually found by a user using the code. In this paper, instead of searching for problems, the uvm_coreservice_t class was chosen for an examination and usage. It was chosen not at random, but because it has a sorted background and limited value as implemented.

VII. UVM Core Services

In UVM 1.2 a new concept was introduced to the UVM, "coreservices". This class was advertised as a way to programmatically redefine the "core services" of the UVM.

The UVM Core Services are described in the LRM as

> F.4 Core service The UVM core service provides a common point for all central UVM services such as uvm_factory (see 8.3.1), uvm_report_server (see 6.5.1), etc. The service class provides a static ::get (see F.4.1.3), which returns an instance adhering to uvm_coreservice_t. The rest of the set_facility get_facility pairs provide access to internal UVM services.

In the UVM 1.2, the core services contains the functions below. The default implementation managed a factory, a report server, a tr_database and other things.

```
virtual class uvm_coreservice_t;
    pure virtual function uvm_factory get_factory();
    pure virtual function void        set_factory(uvm_factory f);


    pure virtual function uvm_report_server get_report_server();
```

```
    pure virtual function void              set_report_server(uvm_report_server server);

    pure virtual function uvm_tr_database get_default_tr_database();
    pure virtual function void           set_default_tr_database(uvm_tr_database db);
    pure virtual function void set_component_visitor( uvm_visitor#(uvm_component) v);
    pure virtual function uvm_visitor#(uvm_component) get_component_visitor();

    pure virtual function uvm_root get_root();
    local static `UVM_CORESERVICE_TYPE inst;
    static function uvm_coreservice_t get();
      if(inst==null)
        inst=new;
      return inst;
    endfunction
endclass
```

UVM IEEE added other things to manage, including printer, packer comparer and copier defaults. It manages seeding and the resource pool. Core services appear to be growing into things that don't appear to be core services.

```
    pure virtual function int  get_phase_max_ready_to_end();
    pure virtual function void set_phase_max_ready_to_end(int max);

    pure virtual function uvm_printer get_default_printer();
    pure virtual function void        set_default_printer(uvm_printer printer);

    pure virtual function uvm_packer get_default_packer();
    pure virtual function void       set_default_packer(uvm_packer packer);

    pure virtual function uvm_comparer get_default_comparer();
    pure virtual function void         set_default_comparer(uvm_comparer comparer);

    pure virtual function int unsigned get_global_seed();

    pure virtual function uvm_copier get_default_copier();
    pure virtual function void       set_default_copier(uvm_copier copier);

    pure virtual function bit  get_uvm_seeding();
    pure virtual function void set_uvm_seeding(bit enable);

    pure virtual function uvm_resource_pool get_resource_pool();
    pure virtual function void              set_resource_pool (uvm_resource_pool pool);

    pure virtual function int unsigned get_resource_pool_default_precedence();
    pure virtual function void   set_resource_pool_default_precedence(int unsigned precedence);
```

Basics

The UVM 1.2 core services had quite a strange use model which recommended setting a `define and then including the UVM code. Including class based code is a bad idea. The same class included in two different places defines two types which are not assignment compatible. These kinds of bugs are very hard to find. Instead of including class definitions, they should be defined (included) in a package. Then the package is imported in many places. There is only one class type created.

UVM IEEE changed the odd implementation in UVM 1.2 from

```
    local static `UVM_CORESERVICE_TYPE inst;
```

to

```
    local static uvm_coreservice_t inst;
```

uvm_init

In the UVM IEEE, a mysterious function named uvm_init() was added. From the UVM IEEE Reference manual:

> Implementation of uvm_init, as defined in section F.3.1.3 in 1800.2-2017.
> **Note:** The LRM states that subsequent calls to uvm_init after the first are silently ignored, however there are scenarios wherein the implementation breaks this requirement.
> If the core state (see <get_core_state>) is *UVM_CORE_PRE_INIT* when uvm_init, is called, then the library can not determine the appropriate core service. As such, the default core service will be constructed and a fatal message shall be generated.
> If the core state is past *UVM_CORE_PRE_INIT*, and *cs* is a non-null core service instance different than the value passed to the first uvm_init call, then the library will generate a warning message to alert the user that this call to uvm_init is being ignored.

This uvm_init and the related uvm_core state are quite strange. They appear to be used to protect initialization and prevent multiple initializations. The coreservices are getting larger and more complicated.

In addition, UVM IEEE added a set() function and changed get(), adding a call to uvm_init.

```
    static function uvm_coreservice_t get();
        if(inst==null)
            uvm_init(null);
        return inst;
    endfunction // get
    static function void set(uvm_coreservice_t cs);
        inst=cs;
    endfunction
endclass
```

This getter/setter pair is what can cause the uvm_init to be called. See the my_init code below for using uvm_init with a new core service.

Discussion

Implementations are often not pretty things to look at – much like making sausage. The end result is a great tasting sausage, but the process is not really something to think about.

The coreservice in UVM 1.2 did not work as a "replacement using extended classes". It required strange includes and a replacement MACRO definition. It didn't matter, since almost no one actually replaced the core services. In UVM IEEE the strange replacement MACRO is removed, and the core service can be replaced in a "normal" way. But there are still issues. The coreservice class can redefine a factory. That factory must be created very early, since the factory registration for uvm_components and uvm_objects uses SystemVerilog static initializers to initialize themselves. This static initialization happens very early in the simulation cycle.

The code to create and initialize a new core service is below in "my_init". It may look simple, but it was quite difficult to get right, and to get working. This is not a simple or easy-to-use process. Documentation and examples will help, as each vendor is expected to supply.

What is a Core Service anyhow?

A core service sounds like a very important part of the UVM. A core operation or central functionality. For example, the report server, the factory and configuration database all are core operations and central functionality in the UVM.

In the UVM IEEE coreservice the factory and the report server can be overridden and replaced. There doesn't appear to be any such capability for the configuration database. Other functions are available to help manage services, but these are minor.

In the authors many years working with the UVM and UVM users, there has never been an occasion for replacing any of these functions from the distributed UVM. The value of these replacements is yet to be proven.

Replacing the factory

Replacing the factory is quite tricky and difficult. If a factory is built and installed in the core services AFTER the UVM has registered the test bench classes with the default factory, then any future access to the factory will be using an "empty" factory. Static initializers and factory registration are beyond the scope of this paper. But in order to properly install a new core services with a new factory a static initializer will be used.

The code below is a static initializer which initializes the core service replacement. A testbench should execute this initializer before others which may register types with the factory, or otherwise use core services.

The function which constructs the new core services, includes a new factory and a new report server. First construct the core service and initialize it by calling uvm_init. Then construct a report server and "set" the core service. Repeat for the factory.

```
function bit my_init();
  my_coreservice_t   my_coreservice;
  my_report_server_t my_report_server;
  my_factory_t       my_factory;

  my_coreservice = new();
  uvm_init(my_coreservice);

  my_report_server = new("my_report_server");
  my_coreservice.set_report_server(my_report_server);

  my_factory = new();
  my_coreservice.set_factory(my_factory);
  return 1;
endfunction
```

In order to create the new core service, call this initialization function.

```
static bit init_my_coreservice = my_init();
```

The variable "init_my_coreservice" could be named anything. The tricky part is that the side-effect of calling the my_init() function is used to construct and install the new core service. See the appendix for the complete code.

VIII. REPORT SERVER

The report server is a commonly changed class in the UVM. Many users have a preference about how the logfiles, info and warnings are formatted. The report server is a relatively easy way to do that.

The report server can be replaced by extending the uvm_report_server class or the uvm_default_report_server. The uvm_default_report_server is just that – the default implementation. It is easiest to start with it. There are many functions to override (in order to change behavior). One example is to replace the function execute_report_message().

This is a function of medium complexity that causes $display() to be called with a formatted message. Changing this function is one way to get a different format (for example to restrict file names to 12 characters or to print just the message and the filename).

Extend the default report server, replacing execute_report_message by copying the entire function into the new class, and then changing it as desired.

```
class my_report_server_t extends uvm_default_report_server;


  virtual function void execute_report_message(
     uvm_report_message report_message, string composed_message);
     ...
  endfunction
endclass
```

The function execute_report_message takes two arguments. The first (report_message) is a structure which contains all the pertinent information about this message. The second (composed_message) is a string that has already been formatted, usually in compose_report_message. The function compose_report_message() is another possibility for override.

When a message is to be printed, the uvm_report_handler sets some information, and calls the report_server function process_report_message. The function process_report_message sets some information and checks some things and calls compose_report_message to format the message. Then process_report_message calls execute_report_message with both the "RAW (unformatted)" report_message and the string which has been formatted by compile_report_message. In summary, the function process_report_message first composes (formats a message) and then causes it to be printed (or otherwise processed).

Any of the routines in uvm_report_server are candidates for replacement. This paper is replacing execute_report_message.

Execute_report_message is a bit of a beast, coming in at 93 lines of code. (compose_report_message is 64 lines). These are the most likely candidates for replacement in a new report server.

The most interesting part of execute_report_message for formatting a different message is shown below. The override will change the $display to a 'fancy' $display.

```
// DISPLAY action
if(report_message.get_action() & UVM_DISPLAY)
  $display("MY REPORTSERVER %s", composed_message);
```

In order to change the $display into something else, copy the entire contents of execute_report_message and then change the $display line. For example:

```
// DISPLAY action
if(report_message.get_action() & UVM_DISPLAY) begin
    string verbosity;

    case(report_message.get_verbosity())
      UVM_NONE: verbosity = "NONE";
      UVM_LOW: verbosity = "LOW";
      UVM_MEDIUM: verbosity = "MEDIUM";
      UVM_HIGH: verbosity = "HIGH";
      UVM_FULL: verbosity = "FULL";
```

```
            UVM_DEBUG: verbosity = "DEBUG";
            default: verbosity = $sformatf("'d%0d", report_message.get_verbosity());
          endcase

          $display("MY FANCY REPORT SERVER: severity=%s fullname=%s, id=%0s, MSG=%s, \
                filename=%s, line=%0d, context=%s, action=%s, verbosity=%s, object=%p",
                      report_message.get_severity().name(),
                      report_message.get_report_handler().get_full_name(),
                      report_message.get_id(),
                      report_message.get_message(),
                      report_message.get_filename(),
                      report_message.get_line(),
                      report_message.get_context(),
                      uvm_report_handler::format_action(report_message.get_action()),
                      verbosity,
                      report_message.get_report_object());
      end
```

The fancy $display above is not that fancy, but it demonstrates access to all the report message information. That information can be formatted in any desired way. For example, the file name could be softened, and any full paths removed or the message string could be truncated at 12 character for a more columnar display.

IX. USING AND REPLACING THE FACTORY

A new factory will be created using UVM IEEE. It will simply be a clone of the existing factory with additional debug information. The factory replacement follows the same theme as report server. Copy the default, built-in version, and replace functions as desired.

The new factory

```
class my_factory_t extends uvm_default_factory;
  function void print (int all_types=1);
    ...
  endfunction
endclass
```

A collection of drivers that may be used as factory overrides

```
class driver extends uvm_driver#(transaction);
  ...
endclass

class driver2 extends driver;
  `uvm_component_utils(driver2)
  ...
endclass

class driver3 extends driver;
  `uvm_component_utils(driver3)
  ...
endclass

class driver4 extends driver;
  `uvm_component_utils(driver4)
  ...
endclass
```

Setting the factory overrides in the test bench

In the first code smnippet, an instance override is used. The instance override picks out a specific instance to override. In the code snippet below, the a1.d and a2.d driver is overridden by driver4 and driver3 respectively.

```
driver::type_id::set_inst_override(driver4::get_type(), "a1.d", this);
driver::type_id::set_inst_override(driver3::get_type(), "a2.d", this);
```

In this second code snippet, a type override is used. In this usage, any driver class is overridden with a driver2 class.

```
driver::type_id::set_type_override(driver2::get_type(), 1);
```

In the case of a conflict the instance override will win. Replacing the factory seems to have limited value, except for additions to debug messages.

X. USING THE CONFIGURATION DATABASE

The UVM configuration database is not available for override in the core services. It is however powerful, useful and hard to use and debug.

Setting values in the configuration database

Setting values combines a scope, a target or lookup name, a variable name and a value.

```
uvm_config_db#(int)::set(this, "*",      "simple_int", 12);
uvm_config_db#(int)::set(this, "a*.d",   "simple_int", 13);
uvm_config_db#(int)::set(this, "a*.sqr", "simple_int", 14);
```

Getting values from the configuration database

Getting values can be performed in many ways. It is best to keep it simple. In the code snippet below, a driver retrieves an integer setting and reports whether it is found or not.

```
class driver extends uvm_driver#(transaction);
  `uvm_component_utils(driver)

  int simple_int;

  function void build_phase(uvm_phase phase);
    if (!uvm_config_db#(int)::get( this, "", "simple_int", simple_int))
      `uvm_info(get_type_name(), "GET CONFIG FAILED", UVM_MEDIUM)
    else
      `uvm_info(get_type_name(), "GET CONFIG OK", UVM_MEDIUM)
      `uvm_info(get_type_name(), $sformatf("simple_int=%0d", simple_int), UVM_MEDIUM)
  endfunction
  ...
endclass
```

Configuration Database Update in UVM IEEE

The configuration database did not get too many changes with the UVM IEEE. It is notoriously hard to debug configuration database mistakes, and surprising that some improvement for debug was not added to the release. Suggested below are two debug improvements. The first one is perhaps hard – it changes the API. The second one seems easier, since it only requires simple instrumentation [3].

Additional arguments to lookup_name

Previous work with the configuration database pointed to a need for better debug generally [3]. A new API, with the calling scope, file and line number will help debug.

```
function uvm_resource_types::rsrc_q_t lookup_name(string scope = "",
                                                  string name,
                                                  uvm_resource_base type_handle = null,
                                                  bit rpterr = 1,
                                                  input uvm_object CALLING_CONTEXT = null,
                                                  input string FILE = "",
                                                  input int    LINE = 0
                                                  );
```

There are many ways to improve debug. Providing the calling scope, the file and line number are easy. These "location" arguments will help with debug, reporting where the original set or get is located. There are a family of configuration database functions which will also need to be changed.

Instrumenting lookup_name

Another area which would benefit from improvement is the implementation of the routine lookup_name(). This routine makes many decisions about what does and does not match. Those decisions are important to understand why a configuration lookup went wrong. For example:

- Does an entry in the name map exist with the specified name? If not, then return.
- Does the name exist, but the queue is empty? If yes, then return.
- Is the type handle null?
- Does the non-null type handle match? In either case print the name as a tracking mechanism.
- Does the scope match? In either case, print the scope name.

Annotating or otherwise instrumenting the routine could provide better debug.

UVM IEEE Changes to lookup_name

The function lookup_name existed in 1.1d. When it moved from 1.1d to 1.2 random stability improvements were added.

```
// ensure rand stability during lookup
begin
  process p = process::self();
  string s;
  if(p!=null) s=p.get_randstate();
  q=new();
  if(p!=null) p.set_randstate(s);
end
```

But no improvement to debug.

When lookup_name moved from 1.2 to IEEE improvements and cleanup were added in the way that the regular expression matching was called (uvm_re_match and uvm_is_match). But no functionality was changed, and no improvement to debug was introduced.

XI. USING THE NEW CORE SERVICES

Once the new core services are installed and in place, they are naturally used by the remainder of the UVM.

The coreservice, factory and report server are constructed, initialized and "set" in a static initializer. See my_init and the static initializer code in the appendix.

Once the coreservice is built, the UVM naturally uses those new services. There is no other programming required. Using the new coreservice is easy, only requiring the $cast for the override assignment. The code below fetches the coreservice using get() and then calls print_topology on the UVM ROOT, and then calls print on the factory.

```
task run_phase(uvm_phase phase);
  my_coreservice_t my_coreservice;

  $cast(my_coreservice, my_coreservice_t::get());

  my_coreservice.get_root().print_topology();
  my_coreservice.get_factory().print();
```

XII. CONCLUSION

The UVM IEEE is a shiny new thing. The reader may be interested to use it. The authors can recommend that it is ready to be used. There is no significant benefit that has been discovered to recommend using UVM IEEE over UVM 1.2 or UVM 1.1d. There are plusses and minuses. What is clear is that the UVM is here to stay and should be adopted as part of each verification team's embrace of new techniques and technologies. Likely the final choice on which version to choose may hinge on legacy IP, a partner's preference, or the recommendation of the vendor-of-choice.

What should be remembered, is that staying away from using all the details of the UVM is a wise decision for debug and portability. Further, improvements in the future UVM releases come from the hard work of the various members of the working group. The working group would appreciate and benefit from new, energetic open source coding experts to help develop and test the next set of new features.

The UVM 1.2 introduced a class with suspect value – the uvm_coreservices. The UVM IEEE further added to the functionality captured in that service class. In this paper the factory was overridden, as suggested by the core services. Overriding the factory, effectively putting in a better factory has never been a problem with any user we have worked with. The core service does not provide a way to override the configuration database. Certainly the configuration database is a core service. The UVM 1.2 implementation did not really work as designed. The UVM IEEE does work as designed, but there are timing issues. A static initializer must be used. In conclusion, the core services class is an unneeded and unnecessarily complex class. The idea of a core service and replacing functionality is a fine idea. In conversation with users this subject has never come up. If it is needed, it is a very low priority. Instead, what always comes up is better debug, simplifications and stability.

Source code is available for all coding discussed. Please contact the authors.

XIII. REFERENCES

[1] Steve McConnell, Code Complete: A Practical Handbook of Software Construction, Second Edition 2nd Edition
[2] SystemVerilog Releases, https://en.wikipedia.org/wiki/SystemVerilog
[3] Rich Edelman, Dirk Hansen, "Go Figure – UVM Configure: The Good, The Bad, The Debug" DVCON Europe 2016.
[4] 1800-2017 IEEE Standard for SystemVerilog, https://ieeexplore.ieee.org/document/8299595
[5] UVM 1.1d, https://www.accellera.org/downloads/standards/uvm
[6] UVM 1.2, https://www.accellera.org/downloads/standards/uvm
[7] UVM IEEE example implementation, https://www.accellera.org/downloads/standards/uvm
[8] UVM IEEE Language Reference Manual, https://ieeexplore.ieee.org/document/7932212

XIV. APPENDIX

A simple example that uses the factory, the configuration database, and installs a new core service.

```
import uvm_pkg::*;
`include "uvm_macros.svh"

import my_coreservice_pkg::*;

class transaction extends uvm_sequence_item;
  `uvm_object_utils(transaction)

  int data;
  rand int duration;

  constraint value {
    duration > 1;
    duration < 10;
  };

  function new(string name = "transaction");
    super.new(name);
  endfunction

  function string convert2string();
    return $sformatf("[%s] data=%0d", get_type_name(), data);
  endfunction

  function void do_record(uvm_recorder recorder);
    super.do_record(recorder);
    `uvm_record_field("name", get_name())
    `uvm_record_field("data", data)
    `uvm_record_field("duration", duration)
  endfunction
endclass

class my_special_transaction extends transaction;
  `uvm_object_utils(my_special_transaction)

  constraint value2 {
    duration > 3;
    duration < 7;
  }

  function new(string name = "my_special_transaction");
    super.new(name);
  endfunction
endclass

class my_sequence extends uvm_sequence#(transaction);
  `uvm_object_utils(my_sequence)

  transaction t;
  my_special_transaction special_t;
  int LIMIT;

  function new(string name = "my_sequence");
    super.new(name);
  endfunction

  function void do_record(uvm_recorder recorder);
    super.do_record(recorder);
    `uvm_record_field("name", get_name())
    `uvm_record_field("LIMIT", LIMIT)
  endfunction
```

```
  task body();
    int simple_int;

    if (!uvm_config_db#(int)::get( m_sequencer, "", "simple_int", simple_int))
      `uvm_info(get_type_name(), "GET CONFIG FAILED", UVM_MEDIUM)
    else
      `uvm_info(get_type_name(), "GET CONFIG OK", UVM_MEDIUM)
      `uvm_info(get_type_name(), $sformatf("simple_int=%0d", simple_int), UVM_MEDIUM)

    `uvm_info(get_type_name(), "Starting", UVM_MEDIUM)
    for (int i = 0; i < LIMIT; i++) begin
      if ((i%2) == 0) begin
        // even numbers
        special_t = my_special_transaction::type_id::create($sformatf("t%0d", i));
        t = special_t;
      end
      else begin
        // odd numbers
        t = transaction::type_id::create($sformatf("t%0d", i));
      end
      start_item(t);
      t.data = i+1 ;
      if (!t.randomize())
        `uvm_fatal(get_type_name(), "Randomize FAILED")
      finish_item(t);
    end
    `uvm_info(get_type_name(), "Finished", UVM_MEDIUM)
  endtask
endclass

class driver extends uvm_driver#(transaction);
  `uvm_component_utils(driver)

  transaction t;
  int simple_int;

  function new(string name = "driver", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    if (!uvm_config_db#(int)::get( this, "", "simple_int", simple_int))
      `uvm_info(get_type_name(), "GET CONFIG FAILED", UVM_MEDIUM)
    else
      `uvm_info(get_type_name(), "GET CONFIG OK", UVM_MEDIUM)
      `uvm_info(get_type_name(), $sformatf("simple_int=%0d", simple_int), UVM_MEDIUM)
  endfunction

  task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(t);
      `uvm_info(get_type_name(), $sformatf("Got %s", t.convert2string()), UVM_MEDIUM)
      #(t.duration);
      seq_item_port.item_done();
    end
  endtask
endclass

class driver2 extends driver;
  `uvm_component_utils(driver2)

  function new(string name = "driver2", uvm_component parent = null);
    super.new(name, parent);
  endfunction
endclass

class driver3 extends driver;
```

```
  `uvm_component_utils(driver3)

  function new(string name = "driver3", uvm_component parent = null);
    super.new(name, parent);
  endfunction
endclass

class driver4 extends driver;
  `uvm_component_utils(driver4)

  function new(string name = "driver4", uvm_component parent = null);
    super.new(name, parent);
  endfunction
endclass

class agent extends uvm_agent;
  `uvm_component_utils(agent)

  uvm_sequencer#(transaction) sqr;
  driver d;

  function new(string name = "agent", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    sqr = uvm_sequencer#(transaction)::type_id::create("sqr", this);
    d = driver::type_id::create("d", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    d.seq_item_port.connect(sqr.seq_item_export);
  endfunction
endclass

class test extends uvm_test;
  `uvm_component_utils(test)

  agent a1;
  agent a2;

  my_sequence seq1;
  my_sequence seq2;

  function new(string name = "test", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);

    driver::type_id::set_inst_override(driver4::get_type(), "a1.d", this);
    driver::type_id::set_inst_override(driver3::get_type(), "a2.d", this);
    driver::type_id::set_type_override(driver2::get_type(), 1);

    uvm_config_db#(int)::set(this, "*",      "simple_int", 12);
    uvm_config_db#(int)::set(this, "a*.d",   "simple_int", 13);
    uvm_config_db#(int)::set(this, "a*.sqr", "simple_int", 14);

    a1 = agent::type_id::create("a1", this);
    a2 = agent::type_id::create("a2", this);

  endfunction

  task run_phase(uvm_phase phase);
    my_coreservice_t my_coreservice;

    $cast(my_coreservice, my_coreservice_t::get());
```

```
    my_coreservice.get_root().print_topology();
    my_coreservice.get_factory().print();

    phase.raise_objection(this);
    for (int i = 0; i < 4; i++) begin
      fork
        automatic int j = i;
        begin
        seq1 = my_sequence::type_id::create($sformatf("seq1-%0d", j));
        seq1.LIMIT = 25 * (j+1);
        seq1.start(a1.sqr);
        end
      join_none
      fork
        automatic int j = i;
        begin
        seq2 = my_sequence::type_id::create($sformatf("seq2-%0d", j));
        seq2.LIMIT = 25 * (j+1);
        seq2.start(a2.sqr);
        end
      join_none
    end
    wait fork;
    phase.drop_objection(this);
  endtask
endclass

function bit my_init();
  my_coreservice_t   my_coreservice;
  my_report_server_t my_report_server;
  my_factory_t       my_factory;

  my_coreservice = new();
  uvm_init(my_coreservice);

  my_report_server = new("my_report_server");
  my_coreservice.set_report_server(my_report_server);

  my_factory = new();
  my_coreservice.set_factory(my_factory);
  return 1;
endfunction

static bit init_my_coreservice = my_init();

module top();
  initial begin
    run_test();
  end
endmodule
```

XV. APPENDIX II – MY CORESERVICE

my_coreservice_pkg.svh:

```
package my_coreservice_pkg;
  import uvm_pkg::*;
  `include "uvm_macros.svh"
  `include "my_coreservice.svh"
  `include "my_report_server.svh"
  `include "my_factory.svh"
endpackage
```

my_coreservice.svh:

```
class my_coreservice_t extends uvm_coreservice_t;
    local uvm_factory factory;
    ...
endclass
```

my_report_server.svh:

```
class my_report_server_t extends uvm_default_report_server;

  virtual function void execute_report_message(
     uvm_report_message report_message, string composed_message);
     ...
  endfunction
endclass
```

my_factory.svh:

```
class my_factory_t extends uvm_default_factory;
    function void print (int all_types=1);
    ...
    endfunction
endclass
```